CLaSSiC Project
Manuscript CLaSSiC–89–23

March 1989

A Visual Object-Oriented Unification System

Joseph Oliger
Ramani Pichumani
Dulce Ponceleón

Center for Large Scale Scientific Computation
Building 460, Room 313
Stanford University
Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704 0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) Manuscript CLaSSiC-89-23	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Stanford University 6b OFFICE SYMBOL (If applicable) 2E254	7a NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Stanford, CA 94305	7b ADDRESS (City, State, and ZIP Code) Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Office Of Naval Research 8b. OFFICE SYMBOL (If applicable) N00014	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-86-K-0565, N00014-87-K-0384, N00014-82-K-0335
8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
A Visual Object-Oriented Unification System

12 PERSONAL AUTHOR(S)
Joseph Oliger, Ramani Pichumani, Dulce Ponceleon

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Interim	FROM _____ TO _____	1989 March	33

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This report introduces a software design platform which departs from the style of most design tools by acting as an extension to, rather than a replacement for, existing design tools. The ultimate goal of this system is to unify and integrate the various functions provided by text editors, graphics editors, text formatters, hypertext and structured decomposition tools. It uses a very general data structure which can manifest itself in a variety of visual forms while enabling the user to easily create and manipulate the objects it represents. This tool is useful for creating large, general purpose, hierarchically structured programs, data structures, documents and other similar objects. Moreover, this system is designed to allow for a wide variety of functions to be performed on these data objects (such as graph traversal or functional mapping), both from within the environment as well as externally (for example, mapping concurrent computations onto parallel architectures using a

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Joseph Oliger	22b TELEPHONE (Include Area Code) (415) 723-0571 / 22c OFFICE SYMBOL

19. Abstract cont.

post-processor. This system is also considered to be a sprinboard for investigating future directions in software design and information management.

A Visual Object-Oriented Unification System

Abstract

This report introduces a software design platform which departs from the style of most design tools by acting as an extension to, rather than a replacement for, existing design tools. The ultimate goal of this system is to unify and integrate the various functions provided by text editors, graphics editors, text formatters, hypertext and structured decomposition tools. It uses a very general data structure which can manifest itself in a variety of visual forms while enabling the user to easily create and manipulate the objects it represents. This tool is useful for creating large, general purpose, hierachically structured programs, data structures, documents, and other similar objects. Moreover, this system is designed to allow for a wide variety of functions to be performed on these data objects (such as graph traversal or functional mapping), both from within the environment as well as externally (for example, mapping concurrent computations onto parallel architectures using a post-processor). This system is also considered to be a springboard for investigating future directions in software design and information management.

Contents

A Visual Object-Oriented Unification System

Joseph Oliger
Ramani Pichumani
Dulce Ponceleón*

March 16, 1989

1 Introduction

There are many tools in existence today which help programmers design and implement large scale software systems. In addition to the plethora of editors, compilers and debuggers, there are a wide variety of software tools which help programmers design code in a structured and methodical fashion. This report introduces *VOUS* , a software design platform which departs from the usual approach by acting as an extention to, rather than a replacement for, current design techniques.

In conjunction with the main goal of allowing users to organize and process various sources of information in a unified manner, several additional goals were targeted:

- Software should be designed to organize disparate data entities such as source files, picture files, design notes, TEX [1] documents, project

*The authors are in the Center for Large Scale Scientific Computing, Department of Computer Science, Stanford University, and have been supported in this work by the Office of Naval Research under contracts N00014-86-K-0565, N00014-87-K-0384 and N00014-82-K-0335 and the NASA-Ames University Consortium Agreement NCA2-150.

reports, data sets, etc, into a single document that reflects their use in a unified structure. At present, the main reason why there are so many types of documents is to simplify matters for the computer, rather than for the user.

- The number of commands and concepts that need to be mastered in order to receive maximum benefit from the system should be kept to a minimum. However, power and sophistication should not be traded-off for the sake of simplicity. Instead, the power of a system should be masked by its simplicity.

- The user should be supported in the task of writing software by having access to a rich set of resources and a framework for linking them together in a structured fashion.

- Software should accomodate the user by allowing for design strategies that the user is most comfortable with. For instance, some may prefer top-down strategies while others may prefer (or need) bottom-up strategies at times to build on existing software.

- The interface and functionality of the system should be alterable and extensible by the user in a simple and natural manner. This extensibility goal has not been implemented in the current version but is being advanced here as a future goal.

Upon invokation of *VOUS* (pronounced *voo*), the user may notice that it looks and behaves very much like a text editor. In fact, it is a wysiwyg text editor. Users can write programs or compose TEX documents much like they would in an emacs editor (the current version supports a subset of the emacs commands). In addition, users can augment text with pictures, icons, buttons and structured graphical objects. The text and graphics can be decomposed into logical pages with icons or buttons linking them together. Documents can be constructed either in a linear sequence or a structured top-down hierarchical fashion, depending on the complexity and nature of the document. *VOUS* can also be used to design templates which can be used just like HyperCard [2] [3] [4] documents to organize information. The unique feature of *VOUS* is its ability to support various types of objects within a single document. For the purposes of this report,

the term *graph* refers to a collection of nodes and arcs while *graphics* refers to anything visualized on a two-dimensional surface (e.g. graphic arts, computer graphics, etc).

Any large software project involves various stages of design, each with their own composition tools. The first stage is the theoretical analysis and preliminary design stage. This stage usually involves a lot of pencil and paper designs with calculations scribbled into notebooks. Next, flow charts or pseudo code fragments are written to outline the structures and algorithms of the program. This is soon followed by the actual coding and debugging stage. Next, test sets are created and used to check the correctness and robustness of the code. Once the project approaches completion, users manuals are written and technical reports are published. All of these activities could conceivably be incorporated into one environment in which duplication of effort is minimized and each stage is logically induced from the previous stage. Furthermore, the environment itself should provide a flexible, programmable extension mechanism much in the same way that gnuemacs [5] utilizes LISP [6] as a programmable user interface. As computers become more and more powerful, greater degrees of project automation can be introduced. For instance, the initial design notes can be used as a declarative specification for an automatic software application generator. That is, the user could enter a list of specifications and constraints which need to be met and the computer would then generate code that would satisfy the requirements.

It is important to emphasize that the current implementation of *VOUS* is designed as a springboard for future directions in software design. The following list summarizes the functions that are supported in the first release of the system:

- Simple wysiwyg editing.
- *Hierachical text/graphics editing (à la* MacDraw).
- Hyper card simulation.
- Directed graph editing and traversing.
- Generating data structures for other programs.
- Structured decomposition of programs, files, etc.

- PostScript previewing.
- File and directory browsing.

While the system is very functional in its present state, it is intended to be an ongoing research effort in the evolution of software tools and design methodologies of the future.

2 Motivations for *VOUS*

One of the major goals of *VOUS* is to provide a facility for building higher order systems out of existing tools and programs. Since most high order systems are built hierarchically out of smaller systems, it is useful to consider both programs and their data structures as a collection of directed graphs and to specify their operations as traversals of such graphs. One can view this paradigm as a visual mapping of recursive symbolic expressions from LISP onto a visual domain. While this is not the first implementation of a visual, recursive programming environment, it should be noted that *VOUS* represents a general solution to a specific design requirement. The visual domain allows certain operations (by no means all) to be specified in a simpler and more intuitive manner with less rigor needed in their composition. A set of information protocols have also been defined to provide a simple yet powerful mechanism for exchanging data via software portals [7]. In particular, this standard is currently being used to design a system in which three dimensional surfaces can be specified from a Mathematica [8] window, hierarchically connected and viewed on a *VOUS* window, passed to a three dimensional grid generation system and finally sent to a composite adaptive grid solver. The adaptive grid solver will then modify the grids, send the modifications back to the grid generator which sends it data structures back to *VOUS* which in turn displays the state of the new system. Such a system will be very useful for modeling and predicting weather patterns, ocean currents and other dynamic fluid flow computations. The user will be able to query the system and look at contour plots or surface plots as the solution unfolds over time. The data structures generated by *VOUS* can be mapped onto various machine architectures (e.g., mainframes, distributed workstations, parallel data flow machines, etc) by postprocessors

so that the actual computations will be performed in a manner that is transparent to the user.

3 Data Model and Operator Set

The fundamental paradigm of *VOUS* is the traversal of a directed graph whose nodes represent data objects or operators. Just as compilers employ a back-end that operates on an abstract syntax tree representating a program, *VOUS* enables the construction and execution (i.e., traversal) of directed graphs whose semantic meanings can be defined by the user. Hence, a visual computation environment need not be limited to dealing with just abstract syntax trees which represent programs. In fact, visual programming environments have not proved to be as useful as designers had once envisioned. On the other hand, systems such as HyperCard and HyperTalk from Apple [9] have demonstrated that visual environments are quite useful for tasks that are more general than traditional computer programming. *VOUS* basically generalizes these systems by providing a means to visually construct, edit and traverse directed graphs of polymorphic data nodes. Having a visual representation of an abstract polymorphic tree can be very useful for many of the following applications:

1. A general dataflow programming "language" in which a variety of existing tools can be ordered and connected together graphically. This can provide a procedural specification for combining many large programs together and passing information back and forth between them. As mentioned earlier, a program such as Mathematica can be used to manipulate equations of surfaces which can then be visually pieced together by *VOUS* which in turn passes its hierarchical information to a program that actually solves the underlying system.

2. A graphical UNIX "make" facility where files are represented by icon nodes and dependencies are formed by linking icons with arcs. Inner nodes can represent "actions" which are invoked in order to transform dependent files into target files.

3. An "n-dimensional" spreadsheet where polymorphic data cells are formed out of recursive lists rather than just simple arrays. The lists can also represent n-dimensional arrays whose elements can in turn be other lists or arrays.

4. Hypertext [10] [11] [12] [13] and "hyper-card" based systems, where textual information can be linked to external references and sources.

5. Higher-level shell programming. UNIX shell scripts involving programs such as awk (pattern matching and substitution), sed (stream editing), grep, etc can be "mapped" onto a graph much in the same manner that "mapcar" is used on s-expressions in LISP.

6. Creation of complicated data structures and data sets for existing programs. E.g., composing hierachical, composite grid structures for use in computational fluid dynamics. Many data structures can be created more easily in a graphical form than with a data definition language (DDL). With a DDL, the user must not only know the correct syntax for the language, but must also construct error free sentences in that language.

7. Outline editors for writing books or other large publications.

8. Structured painting and drawing programs along the lines of MacDraw.

The underlying data model of *VOUS* is a directed graph consisting of node elements and pointers to link the nodes together. The model is very similar in concept to s-expressions used in LISP. The primary difference is that instead of each element containing a *first* and a *rest* field (car and cdr in LISP), a linked list of pointers is associated with each node. In theory, such a structure can be represented by a simpler s-expression, but it has the advantage is that it yields a more isomorphic relationship between the representation of the data structure and the semantic as well as visual meaning of a graph. This basic data structure enables users to create and visualize very general directed graphs in a very straightforward fashion. While the *implementation* of lists in *VOUS* differs from those in LISP, the execution of lists is essentially compatible. McCarthy [24], in his seminal paper introducing LISP, illustrated how a graphical flowchart representing

any computation with a single entry and a single exit can be transformed into a recursive function of s-expressions. This, in conjunction with the fact that an s-expression can be represented by a graph, provides a basis for a graphical programming environment [25].

VOUS enables the user to visually construct and traverse a directed graph using three basic primitives: *push, pop* and *link*. *Push*, when executed on a leaf node, causes a new context to be created and allows for the creation of a subgraph which is then associated with the leaf. This allows for graphs to be specified in a top down fashion if desired. *Pop* returns to the context which was established prior to the push operation. If no such context exists, then a new one is created. This allows for graphs to be specified in a bottom up fashion. The *link* operation allows any arbitrary leaf node to be connected to any subgraph. Using these three operations, graphs can then be constucted in a top down, bottom up, or combined fashion.

While flowcharting has fallen out of favor for writing structured code (and not without good reasons), the ubiquity of graphical workstations has caused a renewed interest in visual programming. Figure 1 shows a simple example illustrating the common underlying paradigm of all structures represented in the system. In this example, the graph represents the set of dependencies among files comprising a simple program. Alternatively, one could construct a graph consisting of parallel computation nodes where the dependencies are implied by the arcs connecting nodes. In this case, the structure can be viewed mathematically as a partially ordered set which has many properties of interest in the field of graph theory [14].

The graphical structure can also be used as a basis for a object-oriented programming system along the lines of languages such as Self [20]. These types of languages handle classes in a slightly different fashion than Smalltalk [27] [28] [29] style languages. Self allows objects to be "cloned" from one another rather than being "stamped" out of a class specification. This sort of object-oriented style is more intuitive for graphical environments since the graph itself contains enough information for new objects to be cloned, subclassed and combined into new objects. New objects can be derived by adding or overriding parts of existing objects [30]. Multiple inheritance can be achieved by creating links from existing objects and combining them into new objects. In *VOUS*, the notion of classes and objects can be uniformly derived from a single entity, namely, the graph itself.

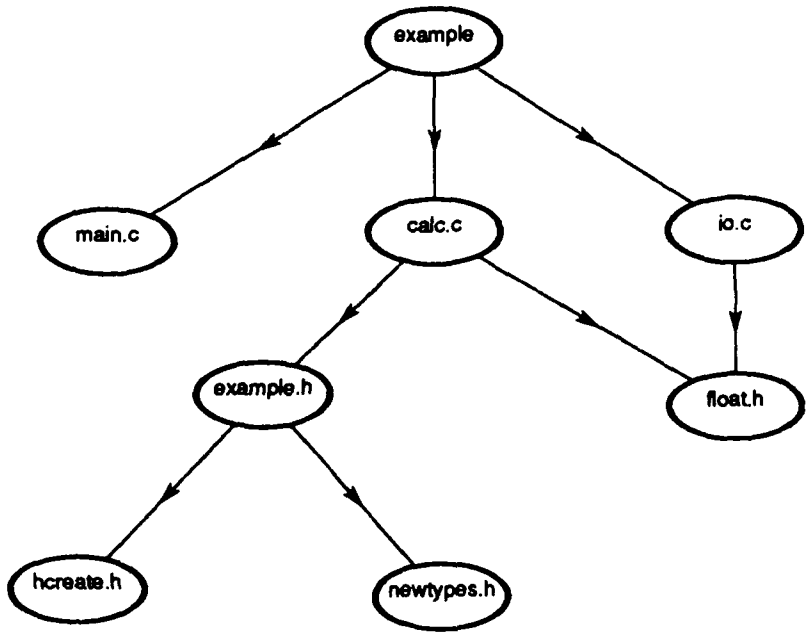

Figure 1: A graph representing file dependencies in a program.

4 Editing Modes

4.1 Graphic Editing Capabilities

VOUS is capable of generating a wide variety of graphical objects. At the lowest level, graphical objects are built on a set of basic primitive operations such as lines, rectangles, circles and curves. Each line, whether straight or curved can have a width parameter which specifies the thickness of the line. The primitive operations can be used to construct drawing paths which can then be filled with a pattern. The primitive operations can also be grouped together to form a structured object and structured objects can in turn be grouped to form larger structures. This process can be repeated indefinitely to create arbitrarily complex graphical entities. The resulting objects can also be cut, pasted and copied like any other object in *VOUS* .

Since *VOUS* utilizes the underlying PostScript [16] [17] [18] [19] imaging model of the windowing system, there is a strong isomorphic relationship between the graphical objects the user constructs and PostScript operands. However, due to the fact that all of the current PostScript servers have virtually no error recovery whatsoever, the user is only allowed, by default, to form syntactically and semantically correct PostScript phrases. Even minor errors in the PostScript source can cause the server to terminate ungracefully. At present, there is a "power user" mode that allows users to define nodes which read in PostScript files. The inclusion of this feature enables *VOUS* to also integrate the functions of a PostScript previewer. However, the implemention of PostScript under NeWS [21] [22] severly limits the range of valid PostScript programs that can be previewed. The authors are hoping to achieve better results on the more recent windowing systems such as the Next Computer's Next Step system which employs a more faithful rendition of the PostScript model. Another primitive object that deserves mention is the bitmap. There is a library of bitmap objects which represent useful icons and symbols which can be copied and pasted into user documents. File and folder icons can be sprinkled anywhere in the document to allow users to view other files, directories and even to execute programs.

There exist a number of predefined structured graphics objects which can be used to simplify the task of constructing certain graphs. These

objects include nodes, arcs, grids and form sheets. Nodes and arcs are useful in constructing the classic structures of trees and flow diagrams. Grids are useful for generating two-dimensional arrays of coordinate values and form sheets are helpful in generating and editing structured data. Figure 2 shows an example of a composite grid created from *VOUS*. Figure 3 shows a screen snapshot taken from a sample session on the system. Another set of predefined objects allows *VOUS* to emulate many of the features of Hypercard and Hypertext (discussed in the next section).

The unique feature of *VOUS* is its ability to synthesize the many different text, graphics and structured objects into one uniform platform (see Figure 4. When the user double clicks on the icon of myfile, the system will invoke the editor on that file. If the file happened to be executable, the system would have executed the file. If the icon was a folder instead, the current directory would have been set to that value. Because *VOUS* supports the inclusion of executable files, documents are not only visually more appealing, but they can exhibit a much more dynamic characteristic than ordinary text documents.

4.2 Other Emulation Modes

Since *VOUS* uses a very generalized list-expression data structure to represent all of its objects, it is capable of simulating a wide range of programs. One of the programs that is easiest to simulate is hypercard. Since nodes can contain objects that can point to (be linked to) other objects and objects can consist of text, graphics and programs, *VOUS* is capable of affecting the behavior of a hypercard user interface. However, the similarity stops at the user interface. Since *VOUS* data structures are modelled after lisp s-expressions, the hypercard simulator can approach the expressiveness of any Turing equivalent programming language. Figure 5 shows an example of a hypercard structure.

All document files created by *VOUS* are stored in ASCII form. Although it is highly discouraged, this allows for the ability to edit the documents using an ordinary text editor. More importantly, it allows documents containing text and graphics to be mailed electronically and edited with most UNIX mail programs. The Berkeley UNIX mail program (also known as

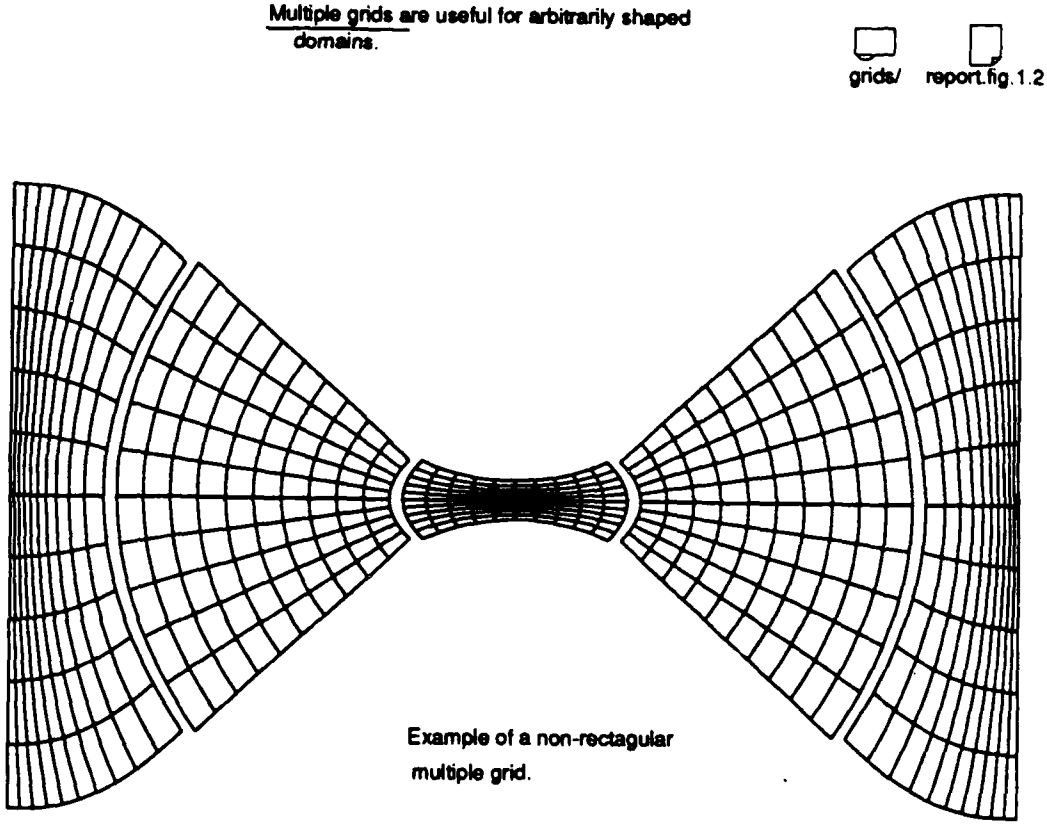

Figure 2: Example of a multiple grid structure created by *VOUS* .

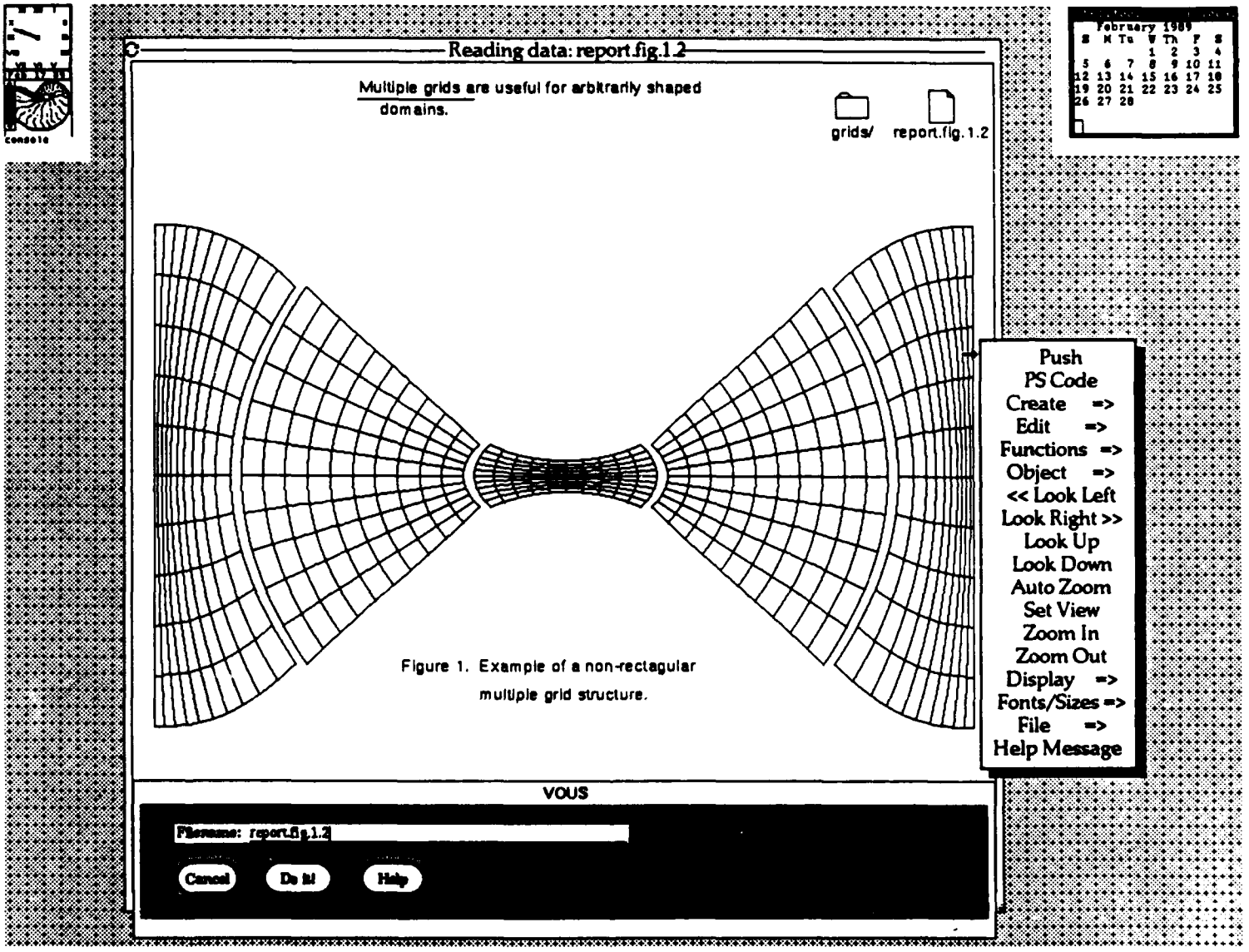

Figure 3: A typical session using *VOUS* to create a multiple grid object.

```
/* This is the usual way of writing C programs: */

#include "myfile.h"
#include "yourfile.h"


/* Now you can write: */
```

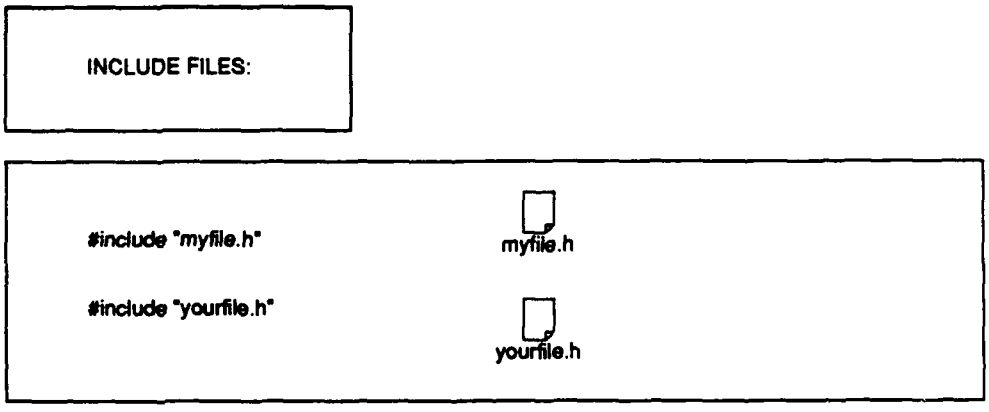

Figure 4: Example of writing source programs with text and graphics. Clicking on file icons invokes the editor on those files. Hence, source programs can contain links to other files.

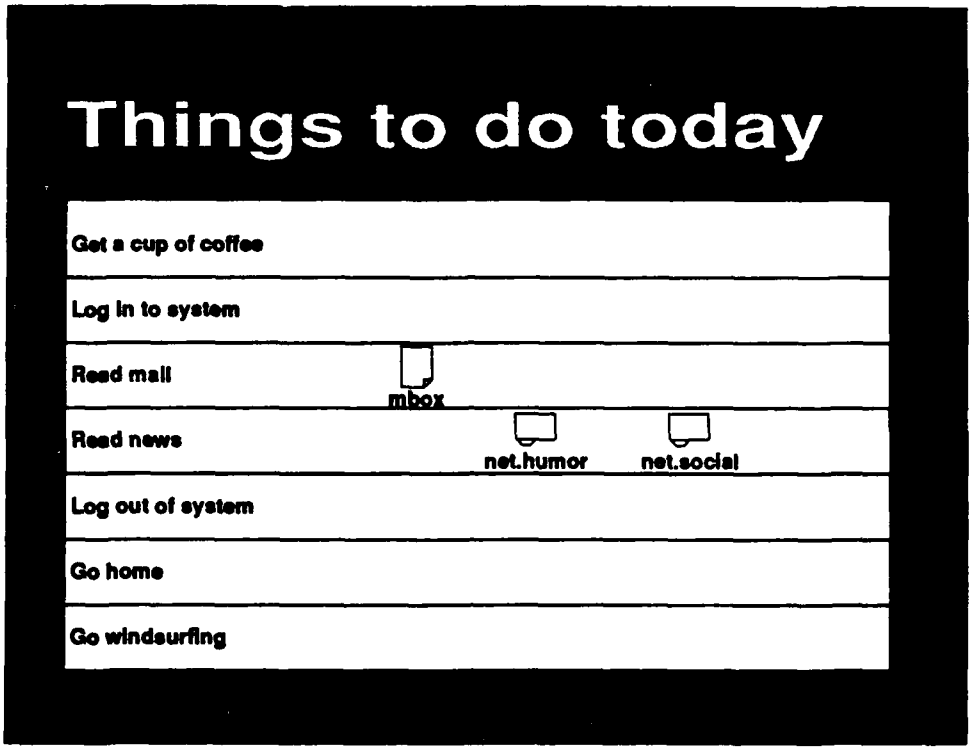

Figure 5: An example of the hyper card simulation capabilities of *VOUS*. Clicking on icons can invoke the editor on files, execute system programs, or traverse links to other pages.

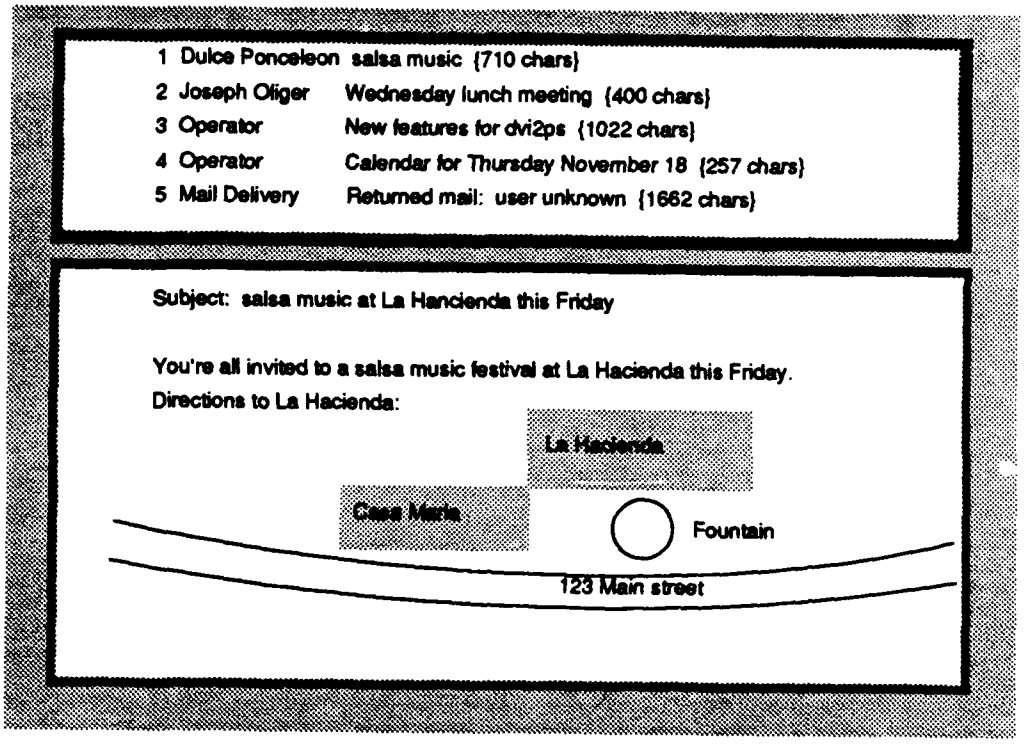

Figure 6: An example of hypermail on *VOUS* .

mailx on System V) has two editors which we call the **e** editor and the **v** editor [23]. The **e** editor is defined by the variable in the .mailrc file called *EDITOR* while the **v** editor is defined by the variable called *VISUAL*. Normally users define the **e** editor to be emacs and the **v** editor to be vi. We have set our **v** editor to be *VOUS* and now have the capability to compose, send and receive documents created by our editor (see Figure 6). The use of this editor enables users to create much richer mail messages since the full power of *VOUS* is available for composing and reading mail. In fact, one can think of this as a "hypermail" application as well.

The hypercard emulation feature is a very convienent way of facilitating a general purpose user interface and data structure entry. Suppose for example, one were to construct an electrical circuit where components are represented by nodes. A node for a transistor might have the appearance depicted in Figure 7.

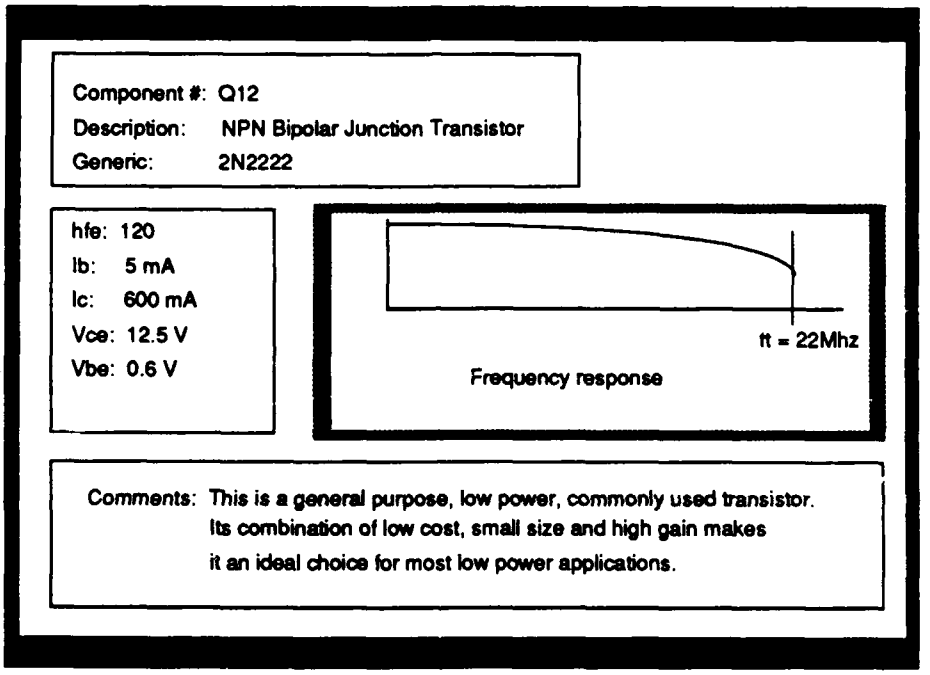

Figure 7: A transistor node and its hyper card data structure. Users can enter and edit parameters or create complicated data structures for other programs (in this example, a circuit simulator).

If the user were to double click on the transistor node, a form sheet would appear containing the various parameters that represent that element. The user can now view and edit parameters of the transistor to see what effects this would have on the circuit response. For instance, simply clicking the cursor in the field called beta (transistor amplification factor) allows the user the change the value very simply and directly. Also note the inclusion of free text descriptions of the node as well as file icons which point to references that may be of use to the user. This method of data entry and editing has many advantages. First of all, the user can see exactly what the parameters are and knows how to edit them without having to learn a set of commands and related syntax requirements. Secondly, the information appears exactly where the user wants it (in this example, right next to the transistor in question). Thirdly, the application builder can easily modify or extend the data structures without having to rewrite the graphical interface program. If the circuit solver reads the data structure produced by the system, when the data field requirements of the solver change, the designer can simply change the form using the system itself. Furthermore, different applications can share data files since the format of *VOUS* data structures is known. That is, it can provide a very consistent medium for transporting structured data from one application to another.

4.3 Desktop emulator

Since *VOUS* provides for file and folder icons, one might logically wonder if it is possible to implement file and directory browsers. The answer is yes - there are two slightly different browsers provided. The first variation performs a directory list of the contents of the current directory (hidden files and all). The second variation performs the same function but removes all the hidden files, relocatable files (ones with .o extensions in UNIX) and emacs auto-save and backup files. This last variation (dubbed the "ls -sane" option by the authors) is probably the most useful form of the directory list feature. Figure 8 shows a sample of the browsing facility.

Files are represented by file icons while directories are represented by folder icons. Double clicking on a folder icon changes the working directory to the selected folder and performs another directory listing. Clicking on

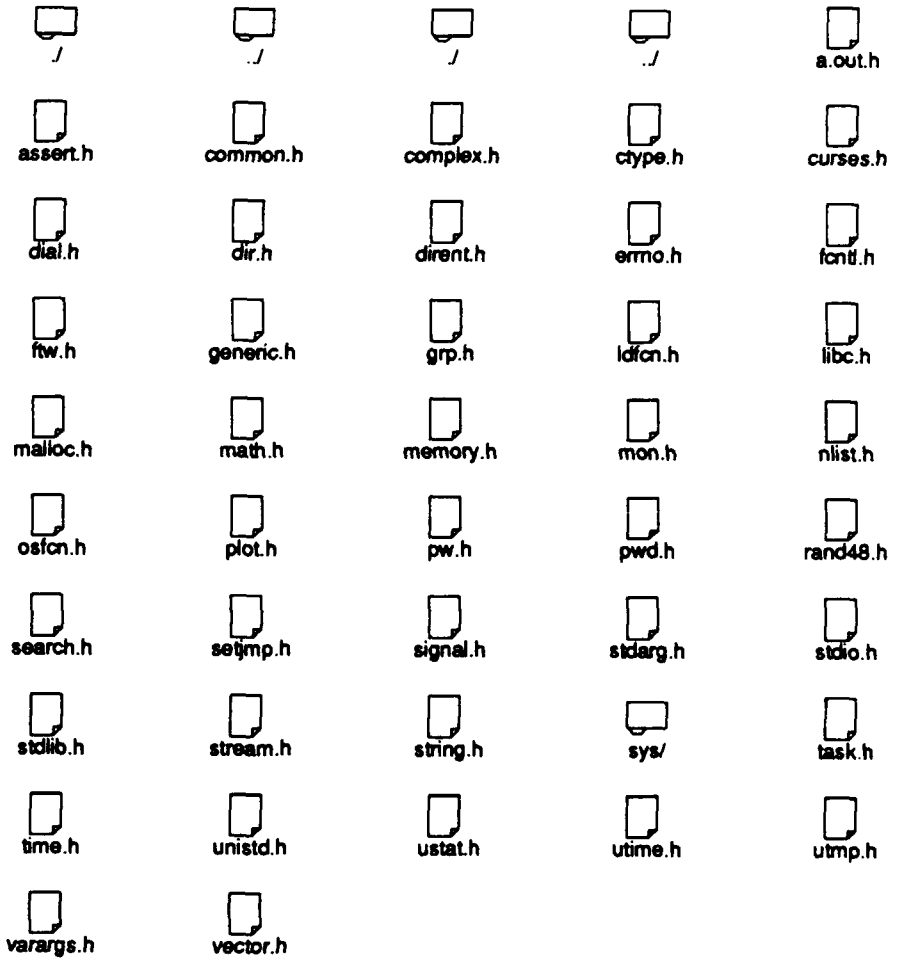

Figure 8: An example of the file and directory browser. Clicking on file icons invokes the editor on those files while clicking on folder icons browses the selected directory.

the "../" folder switches to the parent of the current directory. Clicking on the "./" folder re-executes the list feature on the current directory. Jumping to a random directory can be accomplished by creating a new directory folder and double clicking on it. Double clicking on a non-executable file icon invokes the users text editor on that file. Double clicking on an executable file executes that file in a separate window. Double clicking on a file that has a ".g" extension causes *VOUS* to read the contents of that file, which then becomes the new active file. Unlike the MacIntosh file system, changing the name of a file icon does not change the file name in the operating system. For example, if the file "foo" was changed to "bar", the actual file called "foo" would still exist on the system but would now be inaccessible by *VOUS* . This is because *VOUS* is currently designed to implement *browsing* of files rather than explicit file management. Likewise, in the present implementation, creating a new folder which did not exist in the file system would not actually create a corresponding directory. Even though this would be simple to implement, the authors decided not to include this feature at the present time.

Since the file and folder icons are represented as nodes, whatever operations are allowed on the latter are also allowed on the former. In addition, work is currently underway to perform UNIX commands on an aggregation of files. For instance, searching and replacing in a string in a collection of files is one such example. The goal is to eliminate the need for writing shell scripts to perform common useful functions in the UNIX operating system.

4.4 Text editing features

One of the characteristics shared by every node is that they contain a variable length text string which can be displayed and edited in an emacs-like fashion (i.e., control characters are used to move a cursor forward and backward and characters can be inserted and deleted at the cursor point). The presence of this feature suggested a very simple implementation of a wysiwyg emacs based editor. While the text editing capability is not nearly as powerful as GNU emacs, the ability to handle multiple fonts (including variable width fonts) and variable point sizes makes for a very attractive editing environment for a wide variety of uses (see Figure 9).

Like most wysiwyg editors, the mouse can be used to set the cursor point anywhere on the screen. However, unlike most other editors, the viewed text region can also be zoomed-in, zoomed-out and panned about. One consequence of representing nodal information as a text string is that the entire documented can be viewed and edited as a structured text file if one choses to do so.

The wysiwyg features are easy to implement because *VOUS* is based on the PostScript imaging model which comes with a very rich set of fonts as well as scaling and viewing transformations. In addition, since the list structure of *VOUS* is inherently hierachical, it is also possible to perform outline style editing where nodes can be used to represent sections and subsections with links to text elsewhere in the document. Combining this with all of the other features of *VOUS* provides a multi-faceted composition environment.

It has been said that one of the limitations of wysiwyg editing is that "what you see is all you've got" [26] and no more. This is the reason why programs such as TEX are very popular. TEX is capable of generating very sophisticated typesetting output that is beyond the capability of any known wysiwyg editor. This is because TEX can be thought of as a programming language in addition to a typesetting language. However, the success of LaTeX [26] has shown that virtually the same level of sophistication can be achieved by using a smaller set of macro commands and reserving the more general TEX commands for handling exceptional cases. The goal is to view *VOUS* as a simple, easy to use editor/typesetter but yet, utilize the underlying list expression model to provide very powerful programming control structures. The ultimate goal of *VOUS* is to be as simple to use as a menu driven wysiwyg editor (such as MacWrite) with the text manipulation capability of GNU emacs and the output quality of TeX. Combining the powerful yet orthogonal features of TeX, GNU emacs and MacWrite/MacDraw into one uniform environment results in a highly synergistic and expressive composition tool. While the current implementation of *VOUS* is far from this ideal, it is nonetheless headed in that direction.

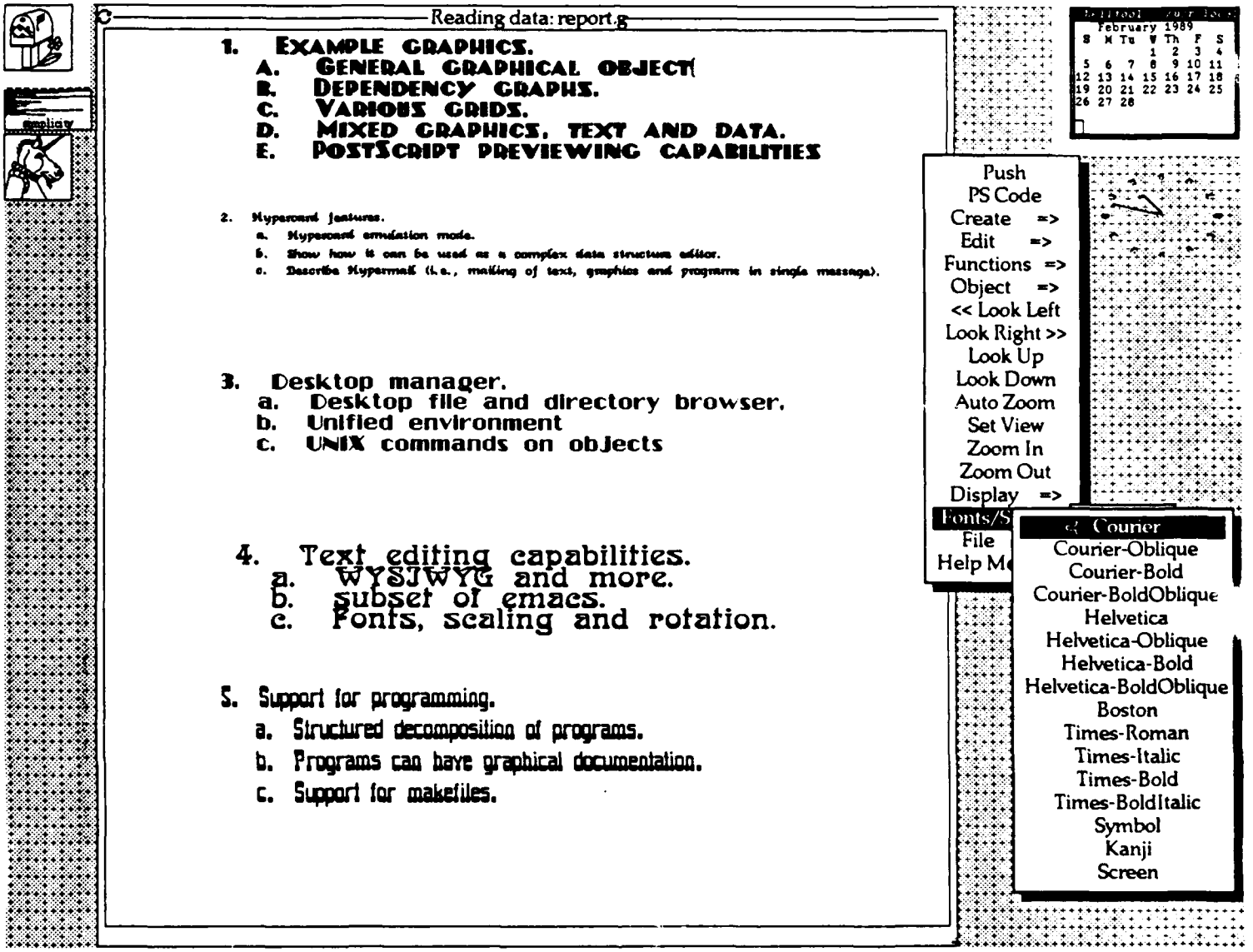

Figure 9: Sample text fonts available on *VOUS* running under NeWS. Not all fonts are available on many PostScript printing devices.

4.5 Support for programming

It should be noted that one of the prime motivations for creating *VOUS* was to facilitate graphical specification and decomposition of large scale programs (in particular, those involving parallel computation algorithms). While the scope of *VOUS* has grown much larger in the course of its design, it nonetheless contains several features which are useful towards this original goal. The simplest feature is file concatenation in which leaves of a graph representing a top down decomposition of a program correspond to collections of program modules which can be concatenated together into a single file. For example, suppose we wish to write a program that consisted of three modules, a data input module, a data processing module and a data output module. One possible strategy is to build a makefile consisting of these three modules plus the main procedure and do a "make" command. Alternatively, we could construct the graph shown in Figure 10.

The node called read would contain the following code fragment (in C):

```
read(args) {

  while(not_empty)
  /* read in data */
  input_data()
  while(process_ready)
  send_data();
  ...
}
```

The process module would look like:

```
process(args) {

  while (data_ready)
  receive_data()
  /* process the data */
```

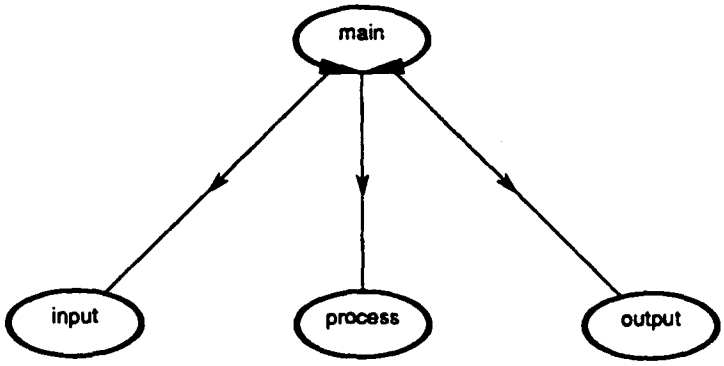

Figure 10: An example of structured program composition.

```
    while(output_ready)
    send_data()
    ...
}
```

And not surprisingly, the output module would look like:

```
output(args) {
    while (data_ready)
    receive_data()
    /* print out the processed data */
    while (not_full)
    output_data()
   ...
}
```

Selecting the file concatenation function would cause the three files to be concatenated together into a single file which could then be compiled. There are also two special types of nodes called "recurance nodes" and "concurrent nodes". If we want to specify that certain computations may done in parallel, we can place each computation node inside a "concur" node. For instance, if we wanted to compute the following line,

$$Result = f_6(f_1, f_2, f_3, f_4, f_5)$$

where each of the computations $f_1 \cdots f_5$ are to be executed in parallel, we could construct a graph as shown in Figure 11.

The concurrent node specifies that the leaf nodes all execute concurrently. Presently, the implementation of concurrency is not addressed in this system. It simply provides a mechanism to specify the presence of concurrency (while preserving the logical structure of the graph) to a back-end program which would read the data structure generated by *VOUS* and implement the concurrency. The "recurrance node" specifies that the child nodes will be executed repetitively in a do or while loop. Again, *VOUS* does not implement the recurance but defers it to the back-end routines. At present, the system merely provides the user with a visually interactive

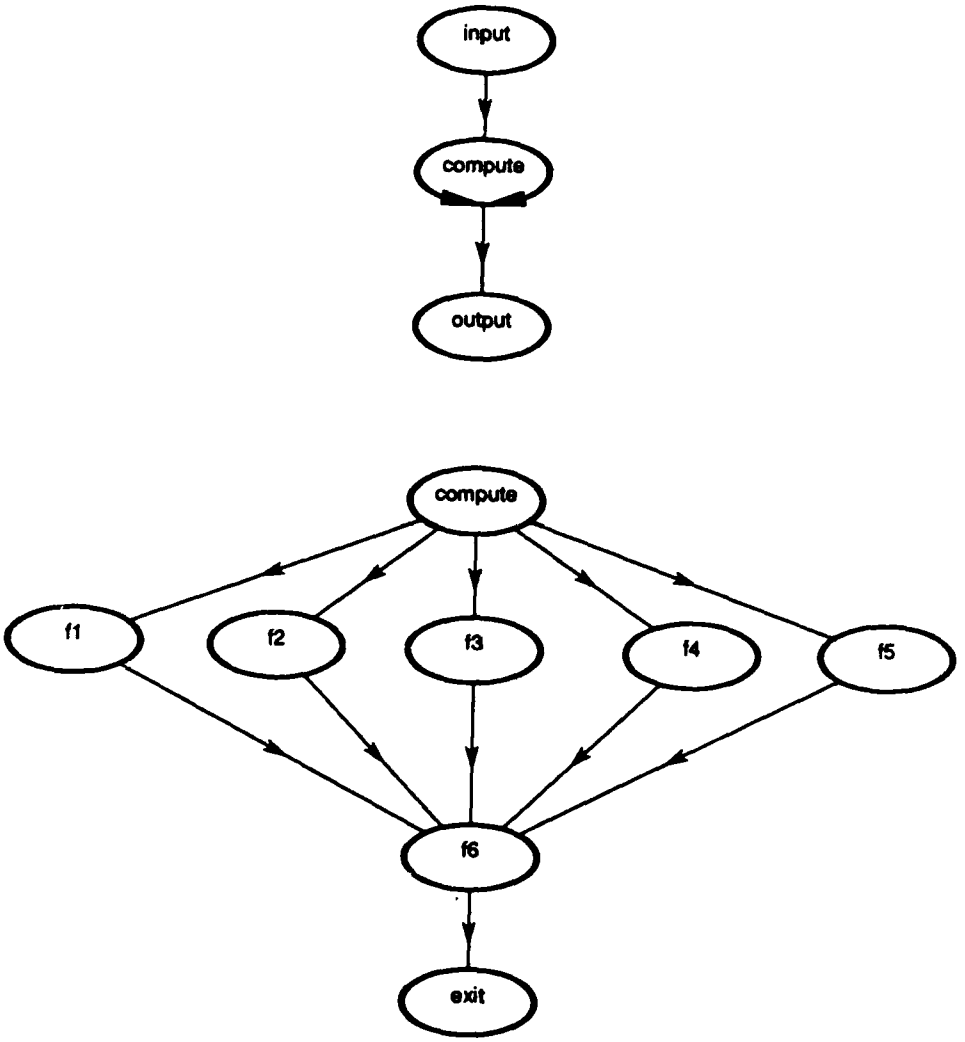

Figure 11: Expressing concurrency in programs.

method of generating and viewing certain properties of algorithms. Thus, the user may specify and map the set of dependencies generated by the system onto any target environment in a machine independent fashion.

VOUS allows for graphs themselves to be decomposed into subgraphs and viewed as separate graphs. It allows subgraphs to be constructed on separate pages with nodes on one page refering to graphs on another page. This allows for the construction of very large graphs from simpler and more manageable ones. Furthermore, it allows graphs to be constructed top-down, bottom-up, or in some combination of the two. If a leaf node of one graph refers to a graph on another page, then double-clicking on that node will cause the system to switch to the refered graph. This process is referred to as "pushing" a node. Likewise, graphs can be "popped" to return to the graph containing the leaf nodes that reference them. In short, graphs can be linked together to form arbitrarily larger and complex graphs. The resulting data structure reflects the linkages in these graphs so that back end functions can view such graphs as a single, monolithic graph.

Some care should be exercised in constructing graphs to ensure that no unintentional cycles are introduced. Intentional cycles can be introduced to implement recursive or cyclical algorithms. However, there are a large class of applications which do not permit cycles to appear in a graph. Some examples of these are directed acyclic graphs (also referred to as DAGs) and trees (probably the most restrictive of commonly used graphs). *VOUS* has a feature which will detect the presence of such cycles and will alert the user to their presence. The goal is to allow the user to select the type of graph to be constructed (DAG, tree, dataflow, etc) and allow the cycle detection feature to forbid the creation of cycles in certain graph modes.

An additional feature to aid program development is the ability to read in makefiles and convert them to into graphs. Work is also underway to perform the reverse function, i.e., convert graphs into makefiles. The goal here is to eliminate the need to create makefiles that must be maintained separately from the program modules. That is to say, makefiles should be automatically induced from the structural composition of their target programs.

VOUS does not attach a semantic significance to the names or the contents of the leaf nodes. Hence, one could also view this as a way to contruct large documents such as books or dissertations. Each leaf could consist of chapters contained in separate files. The leaf nodes in a graph can also rep-

resent other graphs in addition to other files. While technically, this causes a leaf to loose its leaf status (by definition), they can still be viewed as leaves for the purpose of simplifying the view of the graph. In this manner, a book can consist of a graph representing all the chapters as leaf nodes and each leaf node can consist of a graph consisting of section nodes and so on ad infinitum.

4.6 Hardcopies

One of the strongest advantages to using a PostScript based windowing system is the ease with which hardcopies of windows can be generated. Because PostScript utilizes a device independent imaging model, the hardcopy output is significantly superior to that available on even a high resolution workstation display. Virtually any document that can be created and viewed in *VOUS* can be printed on a PostScript based output device. The most commonly available PostScript printers provide anywhere from 300 to 400 dots per inch resolution (dpi) while commercial typesetters provide in excess of 4000 dpi. In addition, *VOUS* allows entire documents to be scaled and translated before printing. For instance, one may choose to have a magnified printout of a small section of a graphical document.

5 The Making of *VOUS*

VOUS is comprised of about 8000 lines of C++ [31] code and an additional 2500 lines of object-oriented PostScript code. It was written on a Sun 3 workstation under the NeWS windowing system. NeWS was selected as the original windowing environment due to the fact that it contains a very powerful imaging model coupled with a very extensible programming platform. The amount of time initially diverted to having to learn object-oriented PostScript turned out to be insignificant compared to the amount of time that was saved by using a very productive windowing system. In fact, the first version of *VOUS* (which was a smaller subset of the current

version) was written in C under the Sunview windowing system. This version contained approximately 7500 lines of source code and produced an 800 Kbyte executable binary. The original features were limited to those pertaining to the construction of directed acyclic graphs. An entirely new editing system was then designed and rewritten for the new windowing system and all the features mentioned above were implemented in a fraction of the original time and resulted in an executable binary of 500 Kbytes. The authors attribute this enhancement in productivity to (1) the extensible object-oriented window system, (2) the PostScript imaging model used by the windowing system, (3) C++, which is not a true object-oriented programming language but nonetheless proved to be a vastly superior programming language to either C or Pascal, and (4) experienced gained from the initial implementation of a directed acyclical graphic editor.

The extensible object oriented windowing system enhances productivity since it is based on an intrepretive graphical language which includes a wide variety of objects to aid the development of application programs. These graphical objects can be used "as is" or can be subclassed, modified and combined with other objects to create new objects. Unlike most windowing systems where there is a *dichotomy* between the window server and the application programs, NeWS application programs tend to be a continuous outgrowth of the windowing system itself. The dichotomy only appears at the interface between the PostScript server and the host language (an interface which behaves like remote procedure calls). The PostScript imaging model allows for a very rich and powerful combination of graphics primitives and support for typesetting. The NeWS windowing system also provides an object-oriented extension to the PostScript language making it conform to the SmallTalk-80 paradigm. The imaging model is made accessible through a very simple yet complete interpretive programming language. The fact that the language is interpretive enables designers to rapidly experiment with the look and feel of the application program. While PostScript does not make a very good general purpose programming language, it does make the task of generating graphical output easier and more enjoyable.

The C++ programming language contributed greatly both to the requirements of the system and the productivity of software writing [32]. While lack of multiple inheritance and dynamic binding were greatly missed, the support for abstract data types (ADT's) and encapsulation of state and behavior of ADT's was very useful. Also, the compiler does quite of bit of

work to "keep the programmer out of trouble" while at the same time, allowing the programmer to write programs that are as equally powerful as those written in C. There are many other object-oriented languages such as Trellis/Owl [33], Eiffel [34] and Objective-C [35] which could have served equally well but were not accessible at the time of implementation.

Since the X windowing system is a far more accepted standard than NeWS, the authors hope to be able to port *VOUS* to this environment in the near future. It is also hoped that the Next Step windowing environment will be much easier to port to since it is based on Display PostScript and Objective-C.

6 Conclusion

While *VOUS* in its current form provides a great deal of functionality, it is nonetheless intended to be the start of an ongoing evolution. With the future incorporation of an extensible programming language, we hope a greater proliferation of user applications will arise. A programmable user interface will allow *VOUS* to move quickly into the domain of end user applications, much in the same fashion that hypertext and hypercard are now being used. Software portals can provide a means for processing and exchanging information over a network of distributed, heterogenous environments consisting of information bases, computational engines, graphics engines and a variety of other servers. The inclusion of these software portals will enable *VOUS* to facilitate better communication with existing programs and provide more transparency, better communication, and synergy of functionality among a wide variety of software design tools and application programs.

References

[1] D.E. Knuth, The TEXBook, Reading, Mass., Addison-Wesley, 1986.

[2] D. Goodman, *The Complete Hypercard Handbook,* New York, Bantam, 1987.

[3] M. Jones, *Hands-On Hypercard,,* New York, J. Wiley, 1988.

[4] G. Harvey, *Understanding Hypercard,* San Francisco, Sybex, 1988.

[5] R.M. Stallman, *GNU Emacs Manual,* Cambridge, Mass., Free Software Foundation, 1986.

[6] P. H. Winston, B. K. P. Horn, *LISP,,* Reading, Mass., Addison-Wesley, 1981.

[7] R. Venkata, *The Standard VOUS Interface,* CLaSSiC Report 89-07, Center for Large Scale Scientific Computing, Department of Computer Science, Stanford University, March 1989.

[8] S. Wolfram, *Mathematica, A System for Doing Mathematics by Computer,* Reading, Mass., Addison-Wesley, 1988.

[9] *Inside Macintosh,* Reading, Mass., Addison-Wesley, 1985.

[10] T. H. Nelson, *The Hypertext,* Proceedings International Documentation Federation, 1965.

[11] T. H. Nelson, *Literary Machines,* edition 87.1, Project Xanadu, 1987.

[12] J. Conklin, *Hypertext: An Introduction and Survey,* IEEE Computer, September, 1987, pp. 17-41.

[13] *Hypertext '87 Papers,* Hypertext '87 Workshop, University of North Carolina, Chapel Hill, 1987.

[14] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *Data Structures and Algorithms,* Reading, Mass, Addison-Wesley, 1983.

[15] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Reading, Mass., Addison-Wesley, 1975.

[16] PostScript Language Reference Manual, Adobe Systems Inc., Addison-Wesley, 1987.

[17] PostScript Language Tutorial and Cookbook, Adobe Systems Inc., Addison Wesley, 1985.

[18] D. A. Holzgang, *Understanding PostScript Programming, Second Edition,* San Francisco, Sybex, 1987.

[19] S. F. Roth, *Real World PostScript,* Reading Mass., Addison-Wesley, 1988.

[20] D. Ungar and R. B. Smith, *Self: The Power of Simplicity,* Stanford, CA, Stanford University, 1988.

[21] *NeWS 1.1 Manual,* Mountain View, CA, Sun Microsystems Incorporated, 1987.

[22] NeWS Technical Overview,, Mountainv View, CA, Sun Microsystems, 1987.

[23] *The UNIX User's Manual, Supplementary Documents,* Section 7, Berkeley, CA, University of California, 1984.

[24] J. McCarthy. *Recursive Functions of Symbolic Expressions, COMM ACM,* 3, 4, April 1960, 184-195.

[25] H. F. Ledgard, M. Marcotty, *A Genealogy of Control Structures,* Communications of the ACM, Volume 18, Number 11, November, 1975.

[26] L. Lamport, LaTeX: A Document Preparation System, Addison-Wesley, 1986.

[27] A Goldberg, *Smalltalk-80,* Reading, Mass., Addison-Wesley, 1984.

[28] T. Kaehler, *A Taste of Smalltalk,* New York, W.W. Norton & Company, 1986.

[29] L. J. Pinson, *An Introduction to Object-Oriented Programming and Smalltalk,* Reading Mass., Addison-Wesley, 1988.

[30] D. C. Halbert, P. D. O'Brien, *Using Types and Inheritance in Object-Oriented Languages,*, Digital Equipment Corporation Technical Report DEC-TR-437, Hudson, MA, 1986.

[31] B. Stroustrup, *The C++ Programming Language,*, Reading, Mass., Addison-Wesley, 1986.

[32] B. Stroustrup, *An Overview of C++,* SIGPLAN Notices, Volume 21, Number 10, October, 1986.

[33] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, C. Wilpolt, *An Introduction to Trellis/Owl,* OOPSLA '86 Proceedings, Association for Computing Machinery, September, 1986.

[34] B. Meyer, *Eiffel: Programming for Reusability and Extendibility,*, SIGPLAN Notices, Volume 22, Number 2, February, 1987.

[35] *Objective-C Reference Manual,* Productivity Products International, Inc., Sandy Hook, CT, 1986.